VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

DETECTION OF API AND ABI COMPATIBILITY IN JAVA

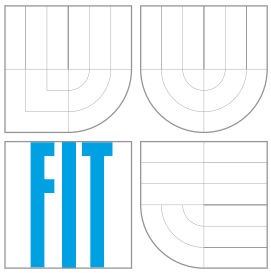
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. TOMÁŠ ROHOVSKÝ

BRNO 2013

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

ZJIŠŤOVÁNÍ API A ABI KOMPATIBILITY V JAVĚ

DETECTION OF API AND ABI COMPATIBILITY IN JAVA

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. TOMÁŠ ROHOVSKÝ

VEDOUCÍ PRÁCE
SUPERVISOR

doc. Dr. Ing. DUŠAN KOLÁŘ,

BRNO 2013

Abstrakt

Tato diplomová práce se zabývá API a ABI kompatibilitou Java knihoven. Jsou popsány typy kompatibility a analyzovány změny API, které vedou k zdrojové či binárni nekompatibilitě. Dále je provedena analýza existujících nástrojů, které provádějí zjišťování nekompatibility. Vhodný nástroj z předchozí analýzy je vybrán a rozšířen. Na základě rozšířeného nástroje je vytvořena serverová aplikace, která poskytuje informace o kompatibilitě sledovaných knihoven.

Abstract

This master's thesis deals with API and ABI compatibility of Java libraries. Types of compatibility are described. API changes causing source and binary incompatibility are analyzed. Furthermore, an analysis of existing tools that detect incompatibility was created. The suitable tool has been chosen from the previously analyzed tools and extended. The extended tool is the base of the server application, which provides information about compatibility of tracked libraries.

Klíčová slova

API, ABI, Java, zpětná kompatibilita, nekompatibilní změny API, vývoj API, Java knihovny, Maven, Hibernate, Spring

Keywords

API, ABI, Java, backward compatibility, incompatible API changes, evolving of API, Java libraries, Maven, Hibernate, Spring

Citace

Tomáš Rohovský: Detection of API and ABI Compatibility in Java, diplomová práce, Brno, FIT VUT v Brně, 2013

Detection of API and ABI Compatibility in Java

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Dušana Koláře

. . . . . .
Tomáš Rohovský
June 3, 2013

Poděkování

Tímto bych chtěl poděkovat mému vedoucímu Dušanu Kolářovi za vedení této diplomové práce. Velké díky patří mému technickému vedoucímu Stanislavu Ochotnickému za cenné připomínky k implementaci knihovny a serverové aplikace, ale také k obsahu a struktuře tohoto dokumentu. Chci poděkovat Martině za kontrolu a úpravu jazykové stránky práce. Obrovské díky patří mé rodině a Ivce za podporu, trpělivost a lásku.

Contents

Chapter 1

Introduction

This master thesis deals with API and ABI compatibility and its detection in Java libraries. Contemporary software development is based on usage of prefabricated components, i.e. libraries. These libraries provide services to the software applications through an API.

Software libraries are evolved as every software product. The evolution of a library may result in incompatible API and ABI changes, which makes complications to the software developers using that library when updating to the new version of the library. These complications can be prevented or minimized by evolving the library API according to particular rules or with the use of tools which detect whether a change causes incompatibility in an API or an ABI. These tools are also very useful for software developers using a library. They can find the newest library release that is compatible with their version or which library is consistent in terms of compatible API. For these purposes, a centralized system providing previously mentioned information will be much more suitable.

The second chapter contains explanation on what an API and an ABI is. It also contains a listing which Java elements are part of an API and types of compatibility are described. Incompatible changes are presented.

The third chapter deals with an analysis of existing solutions. A number of existing solutions detecting incompatible changes is analyzed in several aspects. At the end of the chapter, it is considered whether to extend and reuse one of the existing tools or to start from scratch.

In the fourth chapter, the goal of the thesis is set. Then the current state of the library that will be reused is described and extensions are suggested. An analysis and design of the server application which will provide information about compatibility of tracked libraries follows. The description of the server application use cases is involved. Space and time decisions are also included. It is considered which approach of cohesion will be suitable. The architecture of the server application is designed and a domain model is modeled. Furthermore, it is observed what sources of libraries can be used.

The implementation details of both the library and the server application are described in the fifth chapter. The optimization and refactoring of the library are included. The changes in the model of API are explained as well as parsing of bytecode to it. Improvements of compatibility detection are described and newly supported changes are listed. Deeper implementation details of some more complicated changes are brought. The instantiating of user defined API model classes is depicted. The implementation of the server application continues after an implementation of CLI. Decisions about used technologies were made. Then a description of each layer in the multi-layer architecture of the system follows. It includes these layers: database, domain model, data access, business and presentation.

The sixth chapter contains a design of experiments with the server application and a discussion about the results of the experiments. Based on the results, the system was optimized.

The requirements of the library and the server system are described together with the deployment of the system in the seventh chapter.

At the end of the thesis a summary of what was achieved is described. Possible future improvements are also suggested.

1.1 Motivation

Developers of libraries should know which changes of API are compatible or not. The number of types of incompatible changes is too high to remember them all. A tool which helps developers to keep API compatible or to know what changes affect compatibility is very helpful.

The information about API compatibility is also useful for consumers of the libraries. Some developers of libraries do not provide information on compatibility between the releases of their library. The consumers are forced to the approach of iterative „try to build and change the code if it does not work“ when updating to the new release of the library.

Downloading all newer releases of the library and checking the compatibility with the currently used release of the library is tedious. It is more suitable to have the information about the compatibility stored in one centralized place, so that many users of the library could see it. Such a place can be a web information system.

The system can be based on the top of a tool used by developers of libraries for API compatibility detection. The system should be able to detect compatibility without need of having dependencies of libraries, unlike the tool used by developers of libraries who have access to the dependencies. The reason is that storing of dependencies would take disproportionally more space than a tracked library itself.

Chapter 2

API and ABI Compatibility in Java

2.1 API and ABI

An application programming interface (API) is an interface that a software application implements to enable other software applications to interact with it. API are implemented by applications, libraries and operating systems to define how other software can request services from them. The API itself is abstract, it specifies an interface and does not get involved with implementation details.

An application binary interface (ABI) describes the low-level interface between a computer program and the operating environment.

The difference between an API and an ABI is that an API is source code based while an ABI is binary code based. Therefore the source code using a particular API can be compiled with a software application providing that specific API. While an ABI allows a program from one environment supporting that ABI to run without modifications on any other such environment.

2.2 API Elements and Types of Compatibility

Let assume we have a generic component with an API, which is maintained by one party. Other party, or parties, write the client with using of the component. All parties need to understand which elements are part of the component API and which are part of internal structure of the component. The following convention uses the visibility-limiting feature of Java language to distinguish those Java elements which are considered an API [1]:

API package – a package that contains at least one API class or interface.

API class or interface – a public class or interface in an API package, or a public or protected class or interface member declared in or inherited by some other API class or interface.

API method or constructor – a public or protected method or constructor either declared in or inherited by an API class or interface.

API field – a public or protected field either declared in or inherited by an API class or interface.

There are three main categories of compatibility:

Source
: Source compatibility concerns translating Java source code into class files. In other words, successful compilation is the basic criteria for source compatibility. Source compatibility corresponds to API compatibility.

Binary
: Binary compatibility is preserving the ability to link without error. A change to a type is binary compatible with (equivalently, does not break binary compatibility with) preexisting binaries if preexisting binaries that previously linked without error will continue to link without error [2]. Binary compatibility is equivalent to ABI compatibility.

Behavioral
: Behavioral compatibility includes the semantics of the code that is executed at runtime. The program does with the same inputs the same or an equivalent operation under different versions of libraries [3].

A library has compatible API if it is source and binary compatible. A library has compatible ABI if it is binary compatible. So API compatibility is more strict than ABI compatibility.

2.3 Incompatible changes of API

In this section, the changes causing incompatibility are described. It is evaluated whether they break only source compatibility or both source and binary compatibility. Some of the changes can be incompatible only under certain circumstances depending on the client code. A severity of these changes is set to warning, other changes are considered to be errors. An API type means a class, an interface, an enum, or an annotation type.

2.3.1 Packages

Number	Change	Breaks	Severity
P.1	Add API type	Source	Warning
P.2	Delete API type	Both	Error
P.3	Change public type to non-public	Both	Error
P.4	Change kind of API type	Both	Error

Table 2.1: Incompatible changes of API packages

Change P.1

Import of API types can be done with using of wildcard character. Added API types can be incorrectly resolved, because client's code contains classes with the same name. Let us assume following client's code:

```
import my.classes.*; // contains Colision
import library.classes.*;
```

```
...
    Colision c = new Colision();
...
```

If class `library.classes.Colision` is added to the library, the client will not compile against to the new version of library. The change was considered to be warning because it should not occur when developers are disciplined and do not use wildcard imports.

2.3.2 Interfaces

All interface fields are `public`, `final` and `static`. All interface methods are `public` and `abstract` These facts reduce the count of incompatible changes. Annotation types are a form of interface.

Number	Change	Breaks	Severity
I.1	Add API method	Both	Error
I.2	Delete API method	Both	Error
I.3	Add API field	Both	Error
I.4	Delete API field	Both	Error
I.5	Contract superinterface set (direct or inherited)	Both	Error
I.6	Add API type member	Both	Error
I.7	Delete API type member	Both	Error
I.8	Add member to annotation type with no default value	Source	Error
I.9	Delete member from annotation type (equals to I.2)	Both	Error

Table 2.2: Incompatible changes of API interfaces

Change I.8

Adding a member to annotation type breaks source compatibility. The example of annotation type in library:

```
public @interface Author {
    public String name();
}
```

The annotation can be used in client code as below:

```
@Owner(name="Arthur Dent")
public class House() { ... }
```

If the annotation type

```
@interface Owner {
    String name;
    String nationality;
}
```

is changed by adding of the new member in the library, then client code will not compile. It can be avoided by adding of default value to a new annotation members. So the `email` member will be specified as

```
String email default "earthling";
```

2.3.3 Interface Fields

Number	Change	Breaks	Severity
IF.1	Change type of API field	Both	Error
IF.2	Change value of compile-time constant API field	Both	Error

Table 2.3: Incompatible changes of API interface API fields

Changes IF.2, CF.2, CF.4

Compile-time constant according to [2] must be:

- declared final
- primitive type or String
- initialized within declaration
- initialized with constant expression

The example can be:

```
public static final int ANSWER = 42;
```

The compile-time constant values are always inlined with Java compilers. If the value of a compile-time constant field is changed, then pre-existing clients will not see the new value unless they are recompiled.

2.3.4 Interface Methods

Number	Change	Breaks	Severity
IM.1	Change method name	Both	Error
IM.2	Add or delete formal parameter	Both	Error
IM.3	Change type of a formal parameter	Both	Error
IM.4	Change result type	Both	Error
IM.5	Add checked exceptions thrown	Source	Error
IM.6	Delete checked exceptions thrown	Source	Error
IM.7	Change parameter from variable arity to array type	Source	Error
IM.8	Delete default clause from annotation type member	Both	Error

Table 2.4: Incompatible changes of API interface API methods

2.3.5 Classes

Number	Change	Breaks	Severity
C.1	Add non-abstract and non-static API method (if class is subclassable)	Both	Warning
C.2	Add abstract API method	Both	Error
C.3	Add static API method (if class is subclassable)	Both	Warning
C.4	Delete API method	Both	Error
C.5	Add first API constructor with arguments	Both	Error
C.6	Delete API constructor	Both	Error
C.7	Add API field (if class is subclassable)	Both	Error
C.8	Delete API field	Both	Error
C.9	Contract superinterface set (direct or inherited)	Both	Error
C.10	Contract superclass set (direct or inherited)	Both	Error
C.11	Add API type member (if class is subclassable)	Both	Error
C.12	Delete API type member	Both	Error
C.13	Change non-abstract to abstract	Both	Error
C.14	Change non-final to final	Both	Error
C.15	Rename enum constant	Both	Error
C.16	Delete enum constant	Both	Error

Table 2.5: Incompatible changes of API clases

Changes C.1, C.3

Assume we have the following client's class extending from library's class:

```
public ClientClass extends APIClass {
    protected void hello() {}
}
```

If the following method is added to `APIClass`

```
public void hello() {}
```

then the client code cannot be recompiled because it is not possible to assign weaker access privileges to inherited method.

Similarly if the following method is added to `APIClass`

```
protected static void hello() {}
```

then the client code cannot be recompiled since static method cannot be changed to non-static in the client code.

2.3.6 Class Fields

Number	Change	Breaks	Severity
CF.1	Change type of API field	Both	Error
CF.2	Change value of compile-time constant API field	Both	Error
CF.3	Decrease access from protected to default or private; or from public to protected, default, or private	Both	Error
CF.4	Change final to non-final (if field is static with compile-time constant value)	Both	Error
CF.5	Change non-final to final	Both	Error
CF.6	Change static to non-static	Both	Error
CF.7	Change non-static to static	Both	Error

Table 2.6: Incompatible changes of API class API fields

2.3.7 Class Methods and Constructors

Number	Change	Breaks	Severity
CM.1	Change method name	Both	Error
CM.2	Add or delete formal parameter	Both	Error
CM.3	Change type of a formal parameter	Both	Error
CM.5	Change result type	Both	Error
CM.6	Add checked exceptions thrown	Source	Error
CM.7	Delete checked exceptions thrown	Source	Error
CM.8	Decrease access from protected to default or private; or from public to protected, default, or private	Both	Error
CM.9	Change non-abstract to abstract	Both	Error
CM.10	Change non-final to final	Both	Error
CM.11	Change static to non-static	Both	Error
CM.12	Change non-static to static	Both	Error
CM.13	Change parameter from variable arity to array type	Source	Error

Table 2.7: Incompatible changes of class API methods and API constructors

2.3.8 Class Type Members

The changes for API type members (inner classes, interfaces, enums and annotation types) are basically the same as for API class and interfaces, with the following addition.

Number	Change	Breaks	Severity
CT.1	Decrease access from protected to default or private; or from public to protected, default, or private	Both	Error

Table 2.8: Incompatible changes of API class API type members

2.3.9 Generic Types and Methods

Generic types and methods are these ones which can contain type parameters.

Number	Change	Breaks	Severity
G.1	Add type parameter (if it has some)	Source	Error
G.2	Delete type parameter	Source	Error
G.3	Add, delete, or change type bounds of type parameter	Source	Error

Table 2.9: Incompatible changes of generic API types and API methods

Chapter 3

Analysis of Existing Solutions

The detection of API compatibility between two releases of one Java library is not a new problem. Some solutions of the problem exist. Most of them are discussed in this section. The analysis of existing solutions is important for observing of different approaches to the issue. Another reason is that analyzed tools can be used as the base of the developed system. We will discuss user interfaces, input, output, features, implementation and incompatible API changes covered by the tools. Only missing detection of changes will be described, because the list of supported changes would be very long. Numbers of changes refer to the changes listed in tables in the section 2.3.

3.1 Testing Library Used as Input

All tools were tested with the purpose of discovering supported API changes. They were tested with the same input, which is two versions of the testing library. I have created this library with an effort to cover all API incompatible changes. Every tool has a different output format, thus makes it impossible to create an automatized test. The names of Java elements were chosen so that they would suggest what change the element would be tested for. In the analysis of every tool, an example of the output report containing information about detected API changes will be listed. All outputs were condensed to the size necessary for the demonstration.

3.2 Existing Solutions

3.2.1 Clirr

Clirr is an open-source tool written in Java. It is possible to use Clirr from command line or as Ant target or as Maven plugin. It accepts a set of jar files of the old version and a set of jar files of the new version of Java library as the input. The output of this tool consists of messages reporting about API changes. One message corresponds to one API change. The message contains severity, the code and description of change. An example snippet of output is listed below.

```
ERROR: 7002: evolvingClasses.Class: Method 'public void deleteMethod()' has been
removed
ERROR: 6001: evolvingClasses.Class\$DeleteEnumConstant: Removed field BAR
ERROR: 3003: evolvingClasses.NonFinalToFinal: Added final modifier to class
```

```
ERROR: 2000: evolvingPackages.ChangeGenderCI: Changed from class to interface
ERROR: 1001: evolvingPackages.ChangeToNonPublic: Decreased visibility of class from
public to package
```

The output can be printed in plain text as well as in XML format. Clirr provides an option to include only classes from specified a package and its sub-packages. It does not support intersection with a client project.

Clirr is implemented with using of ASM library for parsing of byte code. An external library dependent class loader is used for loading of classes. In case that some external library class is missing in input jars, the program breaks. It is necessary to link all external libraries by Clirr options.

Clirr does not detect, or detects in wrong way, following incompatible changes:

- Annotations – I.8, IM.8
- Exceptions – IM.5, IM.6, CM.6, CM.7
- Generics – G.1, G.2, G.3
- Method attributes – CM.9, CM.11, CM.12
- Method variable arguments – IM.7, CM.13

3.2.2 Java API Compliance Checker

Java API Compliance Checker is an open-source Perl script. The input of this script is handled by the options for a new and old library release. The input could be specified as a single jar, many jars, an XML descriptor including paths to jars, a directory or an API dump. The output report is stored in an HTML file. This file contains the result of a test (i.e. whether releases are compatible, percentage of compatibility, how many packages, classes, methods are affected), added and removed methods, problems with data types and problems with methods. Every problem has a description of the effect on a client program that uses the API and affected methods of the checked library.

Checker distinguishes between source and binary compatibility. It has options to check only source or binary compatibility. Checker has option to specify the client Java archive that should be checked for portability to the new library version. It is possible to dump a library to specific format, which can be used as an input of the script. The format is hardly readable by human.

The tool is implemented as a single monolithic script. Definitions of messages, rules, generation of HTML output are all in the one file. That approach is not suitable for a future extension and re-usability is very low.

Java API Compliance Checker does not support following incompatibility changes:

- Annotations – I.8, IM.8
- Generics – G.1, G.2, G.3

3.2.3 SigTest

SigTest is a collection of tools written in Java. This collection includes tools to compare APIs and to measure the test coverage of an API. These tools are under GPL license and thus open source. However, the tools are based on Oracle's commercial SigTest tools product. Signature Test tool is the most interesting tool of the collection for our purpose. According to the documentation, SigTest is used to compare the signatures of two different implementations of the same API. This seems to be more general usage than comparison of two versions of the same library, but in principle it is the same.

The Signature Test tool operates from command line. It can generate a signature file, which is a text representation of an API. The format of the signature file is well designed in terms of readability by human and parse-ability by computer. An example is listed below.

```
CLSS public evolvingClasses.Class<%0 extends java.lang.Object>
cons public <init>()
fld protected int deleteField
innr public final static !enum DeleteEnumConstant
meth public void deleteMethod()
supr evolvingClasses.SuperTypeClass
```

Signature files or JAR archives can be used as the input for the tool. It is necessary to include the `rt.jar` library as an input. This library contains all the compiled class files for the base Java Runtime Environment. It also needs all the dependencies of the compared library in input. API of both involved dependencies (JRE and depended libraries) are used for comparison. Comparison without dependencies can be achieved by setting options for excluding or specifying of particular packages. Two possible formats of an output report are available – a human-readable format and a machine-readable format. In the human-readable format, simple API changes are not presented as pair of errors („missing element“ and „added element“) as in the machine-readable format, but as a single API change. The tool has an option to choose between a binary and a source compatibility mode. Another option allows to specify a migration compatibility check mode, which is similar to the message report used in Clirr. The following listing contains an example.

```
Class evolvingClasses.Class
  "E1.2 - API type removed" :
    method public void evolvingClasses.Class.deleteMethod()
Class evolvingClasses.Class\$DeleteEnumConstant
  "E1.2 - API type removed" :
    field public final static evolvingClasses.Class\$DeleteEnumConstant.BAR
Class evolvingClasses.NonFinalToFinal
  "E5.14 - Changing class from non-final to final" :
    CLASS public evolvingClasses.NonFinalToFinal
Class evolvingPackages.ChangeGenderCI
  "E5.12 - Changing class from non-abstract to abstract" :
    CLASS public evolvingPackages.ChangeGenderCI
  "E5.4 - Removing constructor" :
    constructor public evolvingPackages.ChangeGenderCI.<init>()
  ...
Class evolvingPackages.ChangeToNonPublic
  "E1.2 - API type removed" :
    CLASS public evolvingPackages.ChangeToNonPublic
```

The Signature Test tool has many other options, for more detailed description see the documentation [4].

A big advantage of the Signature Test tool is full coverage of API changes that cause incompatibility.

The implementation of SigTest is very large. It has tens of packages and hundreds of classes. It does not use any third party library for parsing the bytecode, but performs that on its own. This makes the code more complex, but perhaps more faster.

3.2.4 Japi-checker

Japi-checker is a small open-source library written in Java. A Maven plugin is build above the library. The plugin is intended to compatibility checking of library developed by user. That means, it is not easily achievable to check compatibility of third party libraries. Checking is done during a Maven verify stage during the building. All the configuration is handled in pom file. The plugin requires presence of a reference version of library in Maven repository as Maven artifact. Artifact details has to be set in the plugin configuration. The library provides two of input – JAR archive and directory with classes. The compatibility report is printed to the standard output. It consist of messages describing changes. Each message includes severity, a class name, in some cases the line of the change and the message text. An example is listed below.

```
[ERROR] evolvingClasses/Class.java: Could not find method deleteMethod in newer
version.
[ERROR] evolvingClasses/Class.java: Could not find field BAR in newer version.
[ERROR] evolvingClasses/methods/Class.java(17): The method nonFinalToFinal has
been made final, this now prevents overriding.
[ERROR] evolvingPackages/ChangeGenderCI.java: The interface evolvingPackages/
ChangeGenderCI has been changed into an class
[ERROR] evolvingPackages/ChangeToNonPublic.java: The visibility of
the evolvingPackages/ChangeToNonPublic class has been changed from PUBLIC to NO_SCOPE
```

Features of the plugin are poor. It is only possible to choose rules which will be used for checking. It does not distinguish source and binary compatibility.

Japi-checker does not detect, or detects in a wrong way, the following incompatible changes:

- Annotations – I.8, IM.8
- Field added – I.3, C.7
- Field attributes - CF.4, CF.5
- Field value – IF.2 CF.2
- Generics – G.1, G.2, G.3
- Method added – I.1, C.1, C.2, C.3
- Method attributes - CM.9, CM.10
- Method variable arguments - IM.7, CM.13

Japi-checker implementation is divided into three parts: models, rules and a checker. Models describe Java elements, rules describe checks of incompatible changes and the checker puts the previous two parts together. ASM 4.0 library is used for parsing of the bytecode to model.

3.3 Evaluation of Existing Solutions

Now, we will choose one of them as the most suitable tool for extending and reusing for the server system. Four tools were analyzed: Clirr, Java API Compliance Checker, SigTest and Japi-checker. All of them are open source, so reusing of all is possible.

Only one tool does not have a console interface - Japi-checker. Most of them have a plain text output, only Java API Compliance Checker provides just an HTML output. SigTest provides the most features, Java API Compliance Checker is also wealthy for features. The most important features are an option to check binary or source compatibility and specifying whether the client should be checked for portability to the new library. SigTest is able to detect the widest range of incompatibilities. At this point, SigTest seems to be a candidate for extending and reusing. However, in the planned server system we also need to compare the libraries independently of dependencies. For this reason, SigTest fails as a candidate. The same applies for Clirr.

Whence it follows that we have only two possibilities - Java API Compliance Checker and Japi-checker. The first named is written in Perl, which is reasonably serious disadvantage, because the server system will be implemented in Java. Another disadvantage is appearance of implementation, it will need a huge refactoring. The current output format as HTML is inconvenient. The previous consideration suggests that the candidate for extending and reusing for our system is Japi-checker.

Chapter 4

Analysis and Design

4.1 Requirements

The purpose of this thesis is to bring the open source tool detecting API backward compatibility which will detect most of incompatibility-introducing changes analyzed in the section 2.3. The tool will allow to choose between binary or source compatibility check. The input of the tool will be JAR archives or directories containing classes. It will be based on Japi-checker.

Furthermore, the server application using the previous tool will be created. It will provide information about incompatibility-introducing changes of library releases. This information will be pre-computed and stored in the database or computed on demand. This approach will be designed to achieve balance between time and space complexity. The database will be optimized to store big amount of data.

4.2 The Library

4.2.1 Current State

Japi-checker consist of the core part and the Maven plugin part. Only the core part is interesting for our purpose. The class diagram of the core part with classes split into packages is on the figure 4.1.

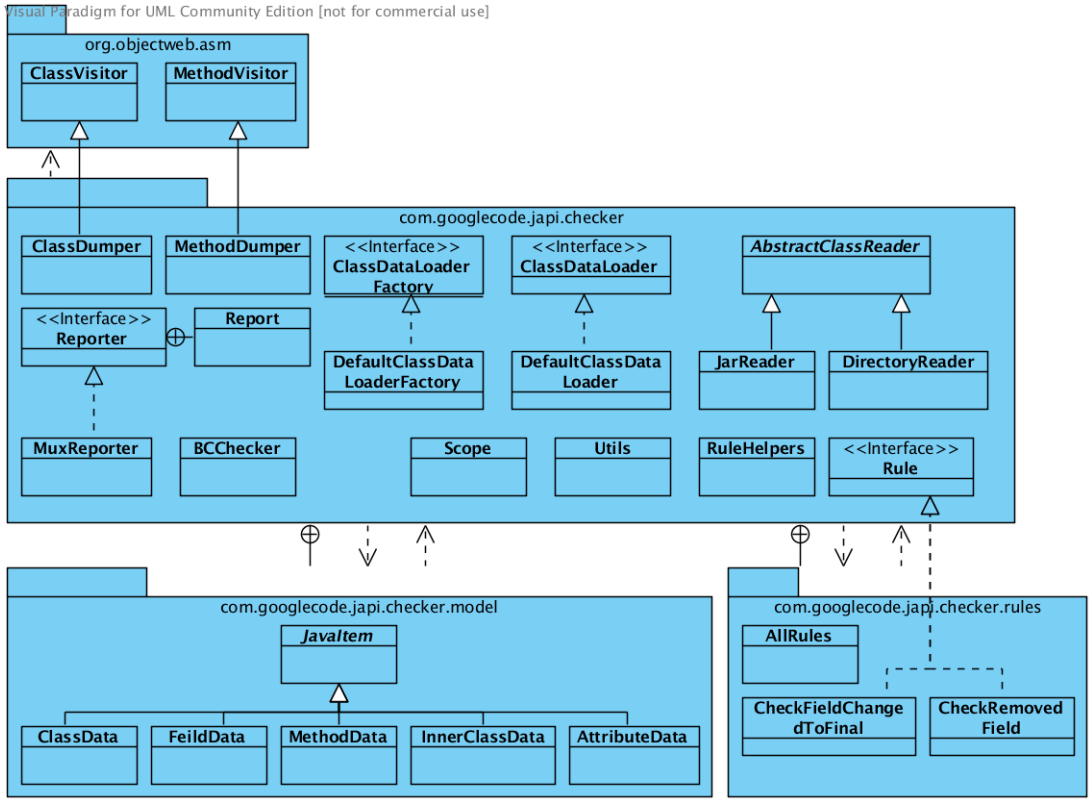


Figure 4.1: Class diagram of Japi-checker.

The main class of the core part is `BCChecker` from the `checker` package. This class creates `DefaultClassDataLoader` for a reference and a new release of a library. This class loads classes. It uses `JarReader` or `DirecotryReader` for reading the classes. The choice of the reader depends on the type of input. The reader uses `ClassDumper` for parsing the bytecode to model. `MethodDumper` is used by `ClassDumper` for parsing of methods from the bytecode. `ClassDumper` and `MethodDumper` inherit from third party ASM library which is used for parsing of the bytecode. When all classes are loaded `BCChecker` checks backward compatibility. Each change is stored to the `Report` class. In the package `model`, there are models of Java elements. The package `rules` contains classes which are used for checking of incompatibilities. Not all classes are shown in the `rules` package in the diagram.

4.2.2 Extensions and Improvements

Japi-checker will be extended for a command line interface, because the current Maven interface is not sufficient. Some developers do not use Maven for building of their libraries. Moreover users of libraries minded to check compatibility cannot easily use Maven interface.

Support to distinguish source and binary compatibility will be implemented. Missing checks described in the section 3.2.4 will be implemented. The rest of checks will be revised and eventually changed. New data structures to support type parameters will be created. The library will be prepared to use in the server application.

4.3 The Server Application

4.3.1 Use Cases

Use cases of the developed application are described in the use case diagram on the figure 4.3.1. Two actors figure in the use case: the user and the administrator. The administrator inherits user's use cases.

The user can display a list of stored libraries and their releases. The user can check API and ABI compatibility of two releases of a specified library. Another user's use case is showing of compatibility overview for the particular library, which contains summary of compatibility between releases in the order as they were released and details of comparisons between releases. The use case for API checking can be extended for an intersection with a client code in the future. Another future extension can be getting an API of a specified release.

The administrator can manage libraries and their releases. The management of libraries comprises adding a library, listing libraries, editing library properties and removal of a library. The management of releases involves adding a release, listing releases, editing release properties and removal of a release.

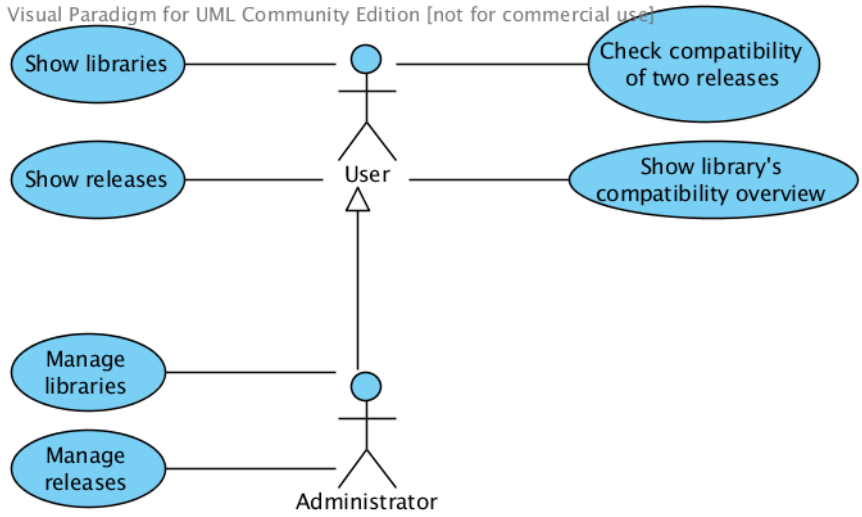


Figure 4.2: Use cases diagram of the system.

4.3.2 Space and Time Decisions

Let us assume we have a library having n releases. The count of all compatibility comparisons between releases is equal to partial permutation

$$P(n, 2) = \frac{n!}{(n-2)!}.$$

For example, in case of library Struts which has 30 prime releases, it would mean to detect the compatibility 870 times. Which is relatively high count. The number of comparisons grows with the number of releases quadratically.

The implication of the previous consideration is storing a large amount of comparison results. However, a complexity of compatibility detection can be balanced between time and space. Some comparisons will be computed on demand. The space complexity can be reduced by storing only backward compatibility comparisons and not storing forward compatibility comparisons, i.e. comparisons needed when downgrading the library release. The reason is that downgrading of libraries is not so frequent. With this optimization will be stored half of all comparisons and space needed for comparisons reduced to approximately 50%.

An algorithm for space and time balancing of comparisons is hard to design at present. We do not know how much space we need for storing the compatibility report and time to its computation. It will be discovered by experimentation with the system (described in the chapter 6). Based on the result of experimentation an algorithm can be designed and implemented afterwards. A vital factor for balancing complexity will be the size of changes between libraries. The minimum needed space will be $n - 1$, where n is a number of releases. The number represents comparisons between releases as they were released. As was previously mentioned, it is required for one of the use cases - getting of library's compatibility overview.

It is important to store data about API of libraries. It is so because of balancing of time and space complexity, but also because of adding of a new library release. Storing a raw library is a waste of space, because only an API is interesting for us. We will store only an API which we will be obtained by parsing of a library.

4.3.3 Cohesion Decisions

There are two approaches how to link the library and the server. The first one is *a strict separation* of the library and the server. The second one is close cohesion of them or rather *an integration* of the library to the server. For both approaches, we will show adding of a release, pre-computation of compatibility, getting a pre-computed result about compatibility and how a computation of the result on demand is proceeded.

In case of strict separation of the library and the server, we need to create an interface for their communication. This interface would be an API dump of release. The API dump will be generated by the library. When adding a release, the server will use the tool to dump the API and store it in the database. A computation of the compatibility will include loading of API dumps from the database and using of them as the input of the library. Getting pre-computed results will only mean querying for the result and presenting it to the user. A computation of the result on demand will be handled as loading of API dumps and using them as the input of the library. The result will be returned to the user.

The integration approach is bringing the library and the server tightly together. There is no need to implement support of API dumping. Adding a release will mean parsing an API by the server and storing that in the database. A compatibility computation will be done by loading data from the database, comparison by the server and storing of it in the database. Getting the pre-computed result is just a database query and retrieval to the user. A computation of the result on demand means loading two APIs from the database and a computation of their compatibility and presenting the result to the user.

The second approach in comparison to the first, can result in relatively duplicitous

development of both applications, although it means better efficiency in terms or time. We will choose the second approach, therefore the integration, mainly to reduce the time needed for an on demand compatibility computation. The business logic of compatibility checking has to be separated, so duplication of development effort will be avoided.

4.3.4 Server Application Architecture

We can divide the system to the back-end and the front-end. The back-end will be a server application which contains a database, a data layer and a business layer. The front-end will contain a presentation layer. The back-end will be only one, whereas there can be more front-ends. The front-end can be realized as a web interface or a CLI client or a Maven plugin or other applications communicating with the back-end by a web service. We will focus on the web interface. Other kinds of front-ends can be created in the future.

4.3.5 Domain Model

A class diagram of the domain model is shown on the figure 4.3. There is a class `Library`, which matches a tracked library. Class `Release` abstracts a release of library. The class implements `ClassDataLoader` interface that gives it a possibility to be used for loading of classes. It contains complete structure of classes modeling API elements. The class `Class` matches API types – a class, an enum, an interface, an annotation. It inherits from library's `ClassData` and adds only `id` required for database. Other API element classes like `Field` of `Method` are not visible for simplification, but they follows principle of extending library's model classes used for `Class`.

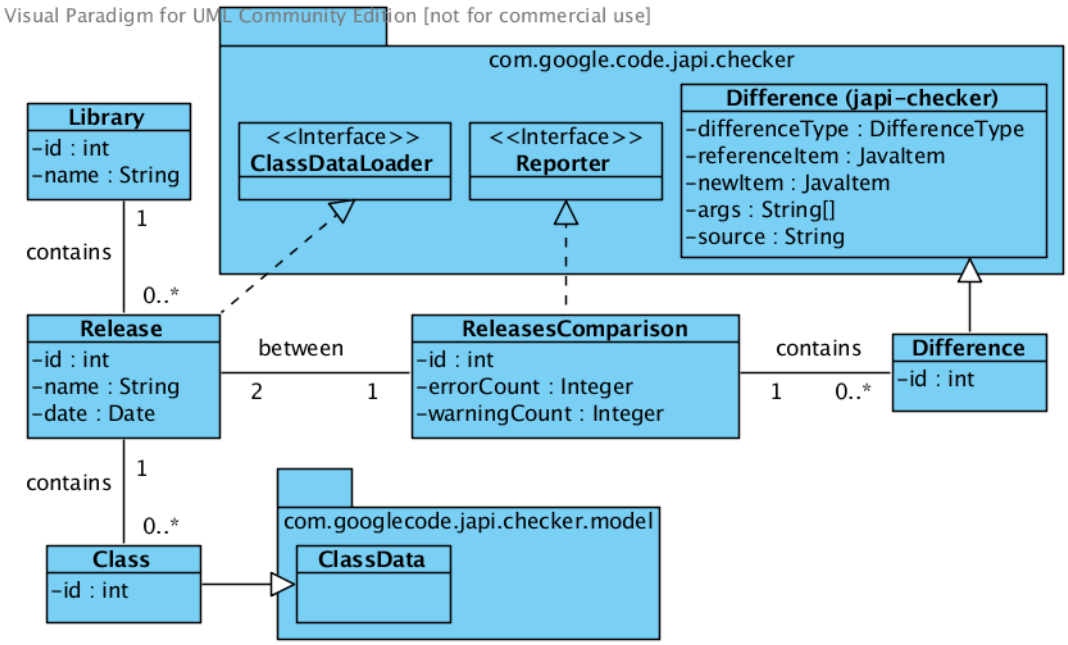


Figure 4.3: Class diagram of the domain model.

Information about a comparison of two releases is stored in `ReleasesComparison`. The class implements library's interface `Reporter` that enables to the class to be used as a container of API changes when checking compatibility. The comparison class is associated

with the class `Difference` that extends from the library's class with same name and it adds only `id`. The library's `Difference` is associated with API elements in which the API change occurred (fields `referenceItem` and `newItem`). The association is not realizable in the final database schema, because it is not possible to refer by one foreign key to more tables that will be created for each API element. This challenge will be solved later (section 5.3.1)

4.3.6 Data Source of Libraries

In order to be the final system usable, it has to contains a lot of data – library releases and comparisons of them. It is necessary to obtain the releases from somewhere. Such a source could be version control system repositories and JAR files placed at library's web pages. The problem of obtaining data from these sources is problematic because of non-uniformity of sources. Another possibility is to obtain these data from Maven Central repository that solves this problem.

Maven Central is a repository of open source Java projects. It contains 678 GB of data, 58 121 of libraries and 484 095 of releases at the time of writing of this document [5]. It was established with the purpose to provide central storage of libraries and to simplify the dependency management when building.

Maven uses following coordinates to identify a certain version of library:

groupId – unique identifier of the project across all projects. It has to follow the package name rules. Example: com.googlecode.japi.checker

artifactId – the name of JAR without version. It correspondents to library in our terminology. Example: japi-checker

version – is the version or of JAR. It correspondents to release in our terminology. Example: 2.0.0

With the coordinates it is easy to define dependencies of libraries, resolving of their dependencies and downloading of them all by building tools like Maven, Ivy or Gradle.

Chapter 5

Implementation

5.1 The Library

5.1.1 Optimization and Refactoring

The library japi-checker has two use cases that lead to two possibly separate business logics. The first one is loading of classes and parsing of API. The second one is checking an API compatibility. In both parts, there are issues with optimization and integration to third party applications (including the information system developed in this thesis).

Thinking about integration of a library in the information system, a problem arises when we want to store parsed API elements. It is because loading of classes and parsing of API was done together with checking an API compatibility in `BCChecker`'s checking method. The loading and parsing has been removed from the checking method and let the user of the library to supply filled model objects on his own. Still using the `DefaultClassDataLoader` created by a `DefaultClassDataLoaderFactory` is the easiest way and thus recommended. That way of loading was also used in japi-checker-cli described in the section 5.2.

One of the issues with checking in the original library was that for each API element all rules are applied regardless to what type of element it is. A type of element was checked in each rule class. That way the element type was checked a lot of times redundantly. To solve this, the process of checking an element type was removed from every rule and all rules applicable for particular element type were put together in classes `ClassRules`, `FieldRules` and `MethodRules`.

A similar problem was redundant checking of visibility in each rule. The solution was also similar, but one thing had to be considered. Although we focus on API checking and thus for public and protected elements, there are cases where elements with lower visibility are needed or interesting. For example checking the increase or decrease of visibility, checking of field's serial version UID or checking of transient keyword. However, the last two make the tool more general than the API checker. The solution was moving the visibility checking from every rule class to one place – in the `ClassRules`, `FieldRules` and `MethodRules`, with respect to non-API rules.

5.1.2 API Model Improvements

Many improvements in the API model were done. The final class diagram is on the figure 5.1

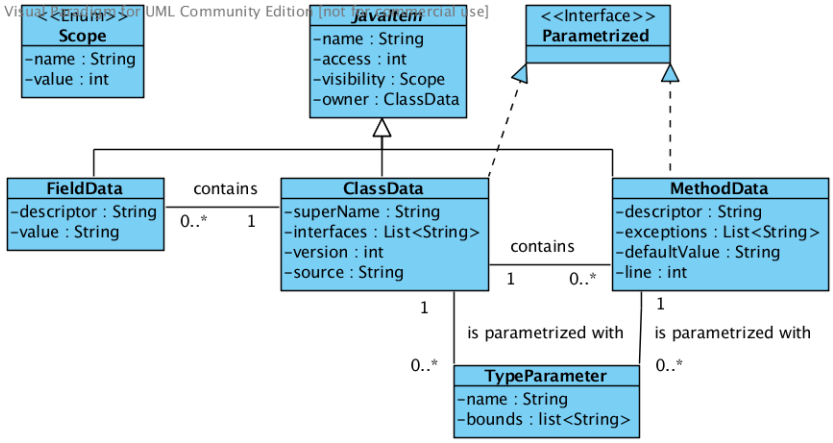


Figure 5.1: Class diagram of API elements.

`JavaItem` abstract class, which serves as the parent for API elements containing visibility, was reduced to contain minimum number of fields. Originally, it contained boolean flags like `isAbstract`, `isInterface`, `isStatic` and others. All these fields were derived from the integer *access* attribute from bytecode. Besides that not all flags are relevant for all API elements (classes inheriting from `JavaItem`), we are losing other information about elements provided by the access attribute. Furthermore, the price of storing one boolean flag is one byte, whether the access (which is integer) needs four bytes, but it keeps all the flags. All these flags were substituted with the `access` field. A desired flag could be evaluated as the following example for a static flag:

```
(access & Opcodes.ACC_STATIC) == Opcodes.ACC_STATIC
```

where `Opcodes` is the interface provided by ASM that defines the JVM opcodes. Methods using this evaluation were added to the class instead of the removed flags.

The field `defaultValue` was added to `MethodData` due to support of annotation type.

The interface `Parametrized` denoting a parametrized type was added. It is implemented by `ClassData` and `MethodData`. The collection `List<TypeParameterData> typePatrameters` storing type parameters was added to these classes. This collection is accessed and filled with methods of the `Parametrized` interface. The class `TypeParameterData` contains information about one type parameter – its name and bounds.

The `signature` field was removed from `ClassData`, `MethodData` and `FieldData`. The `signature` contains the same information as the `descriptor` plus type parameters. As the type parameters are stored in newly created data structures during the parsing of bytecode, which is described in the section 5.1.3, the signature became redundant and thus useless.

5.1.3 Parsing of Bytecode

Several Java libraries have been developed for bytecode manipulation, including BCEL, Javassist and ASM. They enable to modify existing classes before the JVM loads them and

to define new classes at runtime. ASM has already been used in the library, so it was leaved as it was.

Unlike other bytecode manipulation libraries, ASM is focused on simplicity of use and performance. According to the test by [6], it is four times faster than Javassist and almost 8 times faster than BCEL in bytecode generation. However, we are using ASM in an opposite use case – in bytecode parsing.

The ASM library provides two APIs for generating and transforming compiled classes: the core API provides an event based representation of classes, while the tree API provides an object based representation.

The event based API defines a set of possible events and the order in which they must occur. Each event represents an element of a class, such as its header, a field, a method, etc. At the object based model, a class is represented with a tree of objects, each object representing an element of the class. The object based API is build on the top of the event based API [7].

These two APIs can be compared to the Simple API for XML (SAX) and the Document Object Model (DOM) API for XML documents.

The event based API fits best for our purpose, since we do not need to store all objects representing the class. One of the core classes of the API is the `ClassVisitor` abstract class. Each method in this class correspondents to the particular section (class info, field, method, etc.) in the class file. Simple sections are visited with a single method call whose arguments describe their content, and which returns void. Sections whose content can be of arbitrary length and complexity are visited with a initial method call that returns an auxiliary visitor class. This is the case of the `visitAnnotation`, `visitField` and `visitMethod` methods, which return an `AnnotationVisitor`, a `FieldVisitor` and a `MethodVisitor` respectively.

The same principles are used recursively for these auxiliary classes. For example, each method in the `FieldVisitor` abstract class corresponds to the class file substructure of the same name.

The parsing of sections is done by `ClassReader` class. This class parses a byte array conforming to the bytecode and calls the appropriate `visitX` methods of a given class visitor for each class section. The code snippet demonstrating parsing of bytecode is in the following listing.

```
byte[] bytecode;
...
reading of bytecode from the class file
...
ClassDumper visitor = new ClassDumper();
ClassReader reader = new ClassReader(bytecode);
reader.accept(visitor, 0);
```

The japi-checker visitor class `ClassDumper` and auxiliary visitor classes can be seen on the class diagram pictured in figure 5.2. Only essential methods are included in the class diagram.

Every visitor class has a field to store currently parsed data (`clazz, method, item`). This data are filled during the invocation of visit methods.

Originally, there was only `ClassDumper` and `MethodDumper`. I have implemented two auxiliary visitors `AnnotationDumper` and `TypeParameterDumper` to support the annotation

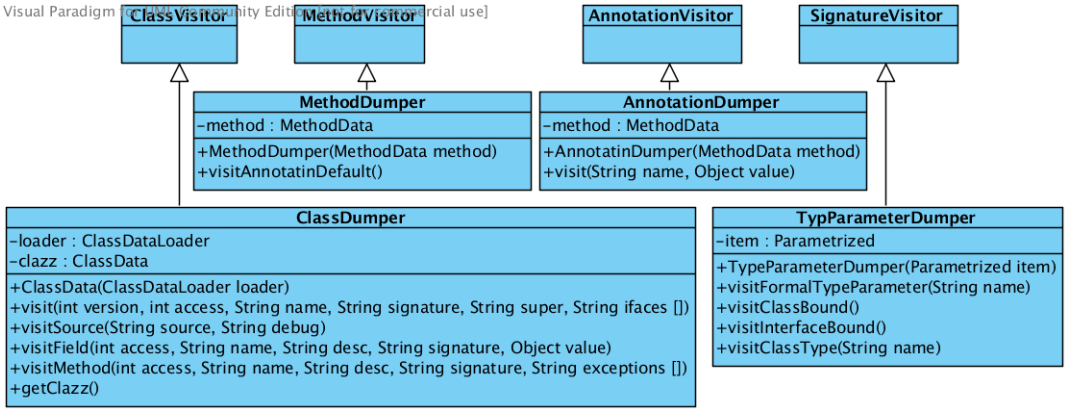


Figure 5.2: Class diagram of visitors.

type and type parameters. Original dumpers had to be changed to support them. A new instance of `AnnotationDumper` is created when `MethodVisitor`'s `visitAnnotationDefault` is called. The annotation dumper is used for obtaining the default value of annotation type's method i.e. annotation's parameter. Here is an example of such a method:

```
public @interface Author {
    String name() default "John Doe";
}
```

`TypeParameterDumper` is used in `ClassDumper`'s `visit` and `visitMethod` methods to parse type parameters of a class and methods. This dumper calls `visitX` methods on the signature of a class or a method. A signature is a descriptor extended for type parameters. The principle of signature reading is similar to bytecode reading. An example of parsing of class's type parameter follows:

```
if (signature != null) {
    TypeParameterDumper = new TypeParameterDumper(clazz);
    Signature reader = new SignatureReader(signature);
    reader.accept();
}
```

A static initialization block (or static initializer) is not a part of API so it was ignored in parsing. It is represented as a method named <clinit> in bytecode.

The naming of Java elements in byte code differs from the naming in source code. Instead of `.`, `/` is used as a separator in bytecode. All names were transformed to a source code naming format before storing to model classes.

The possibility to parse only classes with visibility equal or higher than required – visibility limited parsing – was implemented. The required visibility is defined in `ClassDataLoader`. This feature is intended to save memory. It is useful to reduce the database size in the server system. The reduction is measured in the section 6.1

Another change was replacing of `ClassDumper`'s field `classes`, declared as a map, with the field `clazz`.

5.1.4 Compatibility Detection

The package `rules` contains classes which all implement `Rule` interface. Each class provides implementation of the method

`checkBackwardCompatibility(Reporter reporter, JavaItem r, JavaItem n),`

where one or more API changes are checked. Changes are grouped to rule classes by their similarity. The `reporter` is a container for changes, `r` is the reference API element and `n` is the new one.

Originally, `Reporter` contained a method `report` including these parameters: reference element, new element, description of change (message) and severity. The message was a `String` created by appending literals and variables. Literals together with severity are constants of particular type of change. Let's call them attributes. If a new attribute is added (for example if a change causes only source or also binary incompatibility), the API of the method has to change. This approach is disadvantageous. Moreover, if the same type of change is reported at many places, a duplicate specification of the attribute will be necessary. This led me to creating of the enumeration type `DifferenceType` with attributes of change as fields. Each constant in enumeration already has all of the attributes defined. The enumeration is depicted in the figure 5.3 together with `Difference` and `Severity` classes.

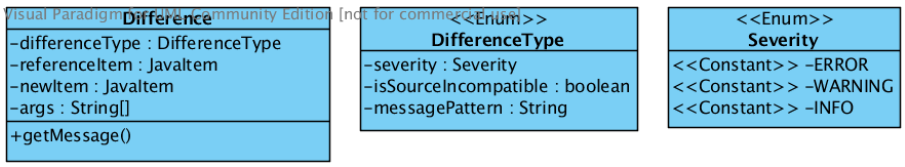


Figure 5.3: Class diagram with difference.

The method `getMessage` uses `messagePattern` and `args` to create a message for particular difference. The message is created by `getMessage` which uses `String.format()` method to return a formatted string. The formatting method applies arguments `args` to the formatting string `messagePattern`. This is the replacement for appending of string literals and variables.

The list of new rule classes with numbers of changes that are covered and was described in the section 2.3 follows:

- ChangeKindOfAPIType – P.4
- CheckAddedField – I.3, C.7
- CheckAddedMethod – I.1, I.8, C.1, C.2, C.3
- CheckFieldChangeToFinal – CF.4, CF.5
- CheckFieldChangedValue – IF.2, CF.2
- CheckMethodChangedToAbstract – CM.9
- CheckMethodDefaultValue – IM.8

- CheckMethodVariableArity – IM.7, CM.13
- CheckTypeParameters – G.1, G.2, G.3
- CheckInheritanceChanges – I.5, C.9, C.10

The rest of the rules was revised and in a few cases corrected. Some of the rules implement really trivial checking like a comparison of a property. Some of the rules contain more advanced logic. These are described below.

CheckAddedMethod

In this class, five incompatible changes are detected. Three of them are changes in class methods. They apply only when it is possible to inherit from the class, i.e. if the class is not final. Namely, these changes are: if an abstract method is added, if a static method is added or a non-abstract and a non-static method is added. The remaining two changes detected by `CheckAddedMethod` relate to interface methods. One of them occurs when an annotation member is added to an annotation type (which is in fact an interface). The other happens when a method is added to an ordinary interface.

CheckTypeParameters

This class contains detection of all three kinds of type parameter changes. That means adding a type parameter, removing a type parameter and changing of type parameter's bounds. The detecting of adding is actually implemented as checking whether a list with type parameters has grown or not. Removing of a type parameter occurs when the size of the type parameter's list has decreased. The detection of type bounds starts with computing of the minimum from the size of reference type parameters and the size of new type parameters. Then every type parameter is compared using an overridden `equals` method. This method first compares the sizes or bounds. If they differ, the type parameters are not equal. If the sizes are equal then each bound is compared.

CheckInheritanceChanges

Former implementation of the japi-checker library did some checking of inheritance. However, the implementation was not correct or rather not total. `CheckInheritanceChanges` contains detection of super class set contraction and super interface set contraction. The big role in this checking is played by the method:

```
List<String> filterAPITypes(ClassDataLoader<?> l, Collection<String> names)
```

which takes the names of types and for each name tries to load an equivalent class by a passed loader. If loading of the class is unsuccessful then the class is from a third party library which means it is a part of API. This class will be returned to other API classes. If loading of the class is successful then its visibility is checked. In case it is higher than the package visibility, the class is meant to be returned.

Super classes are obtained by `ClassData` method `getSuperClasses()`, which returns all names of all super classes. It searches for super classes with the use of `ClassDataLoader`'s method `fromName`. Ideally, the names of super classes are precomputed during the loading of classes, because searching for super classes takes a lot of time. However, in that case it would be necessary to implement some advanced loading technique.

`ClassData` contains also method `getAllInterfaces()`, which retrieves a set of all interfaces. The algorithm has two parts. At first, directly implemented interfaces are added, then all interfaces directly implemented by super classes are added. When we have such a set of directly implemented interfaces, the second part can begin, which is obtaining all indirect interfaces higher in the hierarchy, level by level. It is stated in the following code:

```
// interfaces for processing initialized with direct interfaces
SortedSet<String> inputInterfaces = new TreeSet<String>(allInterfaces);
// newly found indirect interfaces
SortedSet<String> outputInterfaces = new TreeSet<String>();
// obtaining of indirect interfaces
while (inputInterfaces.size() != 0) {
   for (String interfaceName : inputInterfaces) {
        ClassData iface = this.getClassDataLoader().fromName(interfaceName);
        if (iface != null) {
            outputInterfaces.addAll(iface.getInterfaces());
        }
    }
    // preparation for the next iteration
    // remove all interfaces, which have been already collected
    outputInterfaces.removeAll(allInterfaces);
    inputInterfaces.clear();
    // newly found indirect interfaces as a new input
    inputInterfaces.addAll(outputInterfaces);
    // add new interfaces to the result set
    allInterfaces.addAll(outputInterfaces);
    outputInterfaces.clear();
}
```

Having all the previous methods, it is easy to check contraction of a super class set or of an interface set. A method `findAPITypes` is applied for the reference and the new set obtained by `getSuperClasses` or `getAllInterfaces`. Afterwards, it is detected whether the new set contains all elements from the reference set. If positive, nothing happens, otherwise a change is reported.

5.1.5 Instantiating of User Defined API Model Classes

The information system, which will use this library, will use a database to persist API elements. The classes of API elements have to contain an id. The API model classes from this library are not sufficient. There are more options how to solve this problem. We will discuss them now.

The first idea which can come to mind, is to put all values required in the information system's model to the model of the library, so actually there will be only one model of API elements. It is definitely not a good solution, because the library will be influenced by the information system.

One option is to create model classes in the information system, which extend classes from the library and add columns required for the database (at least an id). These model classes will be instantiated and fulfilled from library's API elements received after the pars-

ing. This solution requires traversal of library's API elements and forwarding their values to newly instantiated objects. This way a lot of time and space will be consumed.

Similar solution to the previous one is to create a wrapper class for each API element class from the library, which will contain an instance of the class from the library. In other words composition will be used instead of inheritance. But still a lot of time will be consumed during the traversal over all the objects from the library.

Another option is to instantiate objects from required classes directly in the library. There are two ways how to achieve that. The first one is to use one of the bytecode manipulation libraries described in the section 5.1.3 to modify classes of the library in an information system. The second one is the use of reflection. Specifically to use either:

```
T Class.newInstance()
```

or combination of

```
Constructor<T> Class.getConstructor(Class<?>... parameterTypes)
T Constructor.newInstance(Object... initargs)
```

The method `newInstance()` creates a new instance represented by `Class` object using the default constructor. Method `newInstance(Object... initargs)` uses the constructor represented by `Constructor` object to create and initialize a new instance of the constructor's declaring class, with the specified initialization parameters. This suits us more, because constructors with all required arguments are defined in the library's models.

I have decided to instantiate the required object directly in the library with the use of reflection. Manipulation of an existing class could be good as well and maybe faster but from the point of view that we are creating a library, it is more suitable not to use it. Because users of this library would have to deal with bytecode manipulation library, which requires some knowledge. In case of using reflection they will have a straightforward way to parse API to objects to objects instantiated from required classes.

Fields of `Constructor` type were added to all dumper classes where new classes are instantiated. Setting of these fields was handled in constructors of dumper classes.

5.2 CLI

I have created CLI which is in a separate Maven module. It requires two arguments as input – a reference and a new release to check. It checks binary and source compatibility by default. This behavior can be changed to check only binary compatibility by using `-bin` parameter. This CLI contains two classes `CLIReporter` implementing `Reporter` interface, and `Main` where parameters are processed, libraries are parsed and a report is printed.

5.3 The Server Application

When writing a complex web application, starting from scratch is out of the question. Instead of implementing the persistence layer directly with JDBC, some Object-relational mapping (ORM) framework can serve. Many solutions exist, the most common ones are DataNucleus, EclipseLink, Hibernate, iBatis and OpenJPA. I have decided to use Hibernate as the most widely used Java ORM framework.

Another thing is Java framework for a web application architecture. Enterprise JavaBeans (EJB) or Spring Framework come to consideration here. Both provide transaction management, integration with persistence and security. Both use dependency injection to simplify configuration and integration of heterogeneous systems. EJB is a standard whereas Spring Framework is not. To deploy and run EJB, JEE application server can be used. Alternatively, a standalone container such as OpenEJB can be used. Despite that, Spring can be used in a web server such as Apache Tomcat, which is more lightweight. Because of that, I have chosen Spring Framework.

The last technological consideration is choosing of Model View Controller framework (MVC). Various MVC frameworks such as Struts, JSF and WebWork can be plugged into Spring [8]. It also provides its own solution – Spring MVC. I have chosen Spring MVC as the easiest one to integrate and use with Spring.

The web application structure is divided into the following packages:

- org.fedoraproject.japi.checker.web
  - controller – controllers and forms
  - dao – Data Access Object (DAO) interfaces
    - impl – DAO implementation
  - model – model classes
  - service – service interface and implementation
  - utils – utilities

5.3.1 Database Layer

MySQL database was used in the system. Database scheme is more or less transformation of library model classes and server application model classes. However, some issues arise in the transformation. The ER diagram is on the figure 5.4

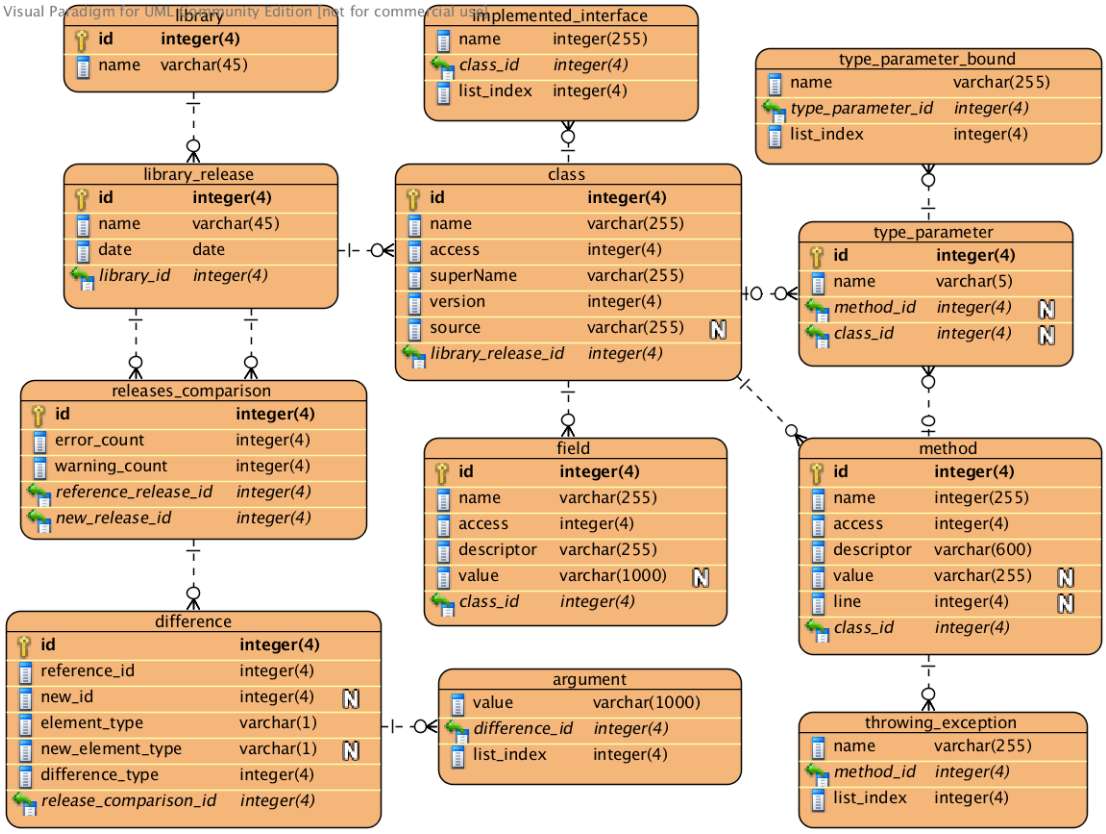


Figure 5.4: Database scheme.

The `Difference` class contains field `referenceItem` and `newItem`. These fields can contain instances of arbitrary classes extending `JavaItem`. The problem is how to transform that matter of fact to a database, because a foreign key can only refer to a single table.

A straight forward solution can be adding foreign keys with possibility to be null for every API element – tables `class`, `field` and `method`. Three times more columns than fields will be needed.

More complicated solution is to create table for `JavaItem` class itself – `java_item`. The foreign keys in `difference` table will refer to the `java_item` table. However this brings additional overheads with composing of object from two tables.

A flag column that will indicate concrete element (table) can be added to the `difference` table. This solution requires smallest interventions to the database model, so it was chosen.

Type `varchar` was used for strings, because it is intended for variable-length strings. Unused space is not padded with white spaces as in case of the `char`. The maximum length of `varchars` was estimated taking into account ordinary maximum lengths and modified as necessary during the testing on the real data.

Indexes were created on the columns in dependence on the queries executed on the database. Namely indexes were created in the `date` and `name` columns of `release` table and in the `name` column of `library` table.

5.3.2 Domain Model Layer

As it was already mentioned, Hibernate framework was used for ORM mapping. A persistent class (entity) has to be connected with a database table and its properties have to be connected with a database column in Hibernate. Framework provides two possible ways to map objects to database tables – using XML or annotations. Most of the domain classes in an information system inherit from library's classes [9]. The properties from library's classes cannot be annotated so XML mapping is the only feasible solution.

We will demonstrate basic mapping on the Library class.

```
<class name="Library" table="library">                    (1)
  <id name="id" type="int">                               (2)
    <column name="id" />
    <generator class="increment" />
  </id>
  <property name="name" type="string">                    (3)
    <column name="name" />
  </property>
  <bag name="releases" inverse="true" lazy="true"         (4)
  fetch="select" cascade="save-update, delete">
    <key>
      <column name="library_id" not-null="true" />
    </key>
    <one-to-many class="Release" />
  </bag>
</class>
```

The declaration of mapping between `Library` class and `library` table is on line 1. Each entity has to have an id defined. Line 2 is doing that. Class's property id is mapped to the column id with an incremental generator defined. The property declaration on line 3 is similar to id. The last and the most interesting declaration is the collection declaration on line 4. The property `releases` declared as a list in the Java class is mapped through the foreign column `library_id` to releases of the library loaded to `Release` class. There, a way is defined how elements of collection are loaded – with `lazy` attribute, fetching strategy with `fetch` and enabling operations to be cascaded to child entities – with `cascade`.

Hibernate requires the presence of a constructor with zero arguments in persistent classes. Therefore against the rule, that a program using a library should not influence the library, zero argument constructors were added to the classes. It is not possible to instantiate these classes with zero constructors, because they are `protected`.

The enum `DifferenceType` is not persistent, but we need an information about what type of difference. Hibernate provides declaration of property as enum type:

```
<property name="differenceType">
    <column name="difference_type"/>
    <type name="org.hibernate.type.EnumType">
        <param name="enumClass">
            com.googlecode.japi.checker.DifferenceType
        </param>
    </type>
</property>
```

Hibernate provides *any* mapping for mapping of one "foreign key" that can refer to one of multiple tables. The `referenceItem` field is mapped as:

```
<any name="referenceItem" meta-type="string" id-type="int" lazy="true">
    <meta-value value="C" class="Class"/>
    <meta-value value="F" class="Field"/>
    <meta-value value="M" class="Method"/>
    <column name="element_type"/>
    <column name="reference_id"/>
</any>
```

Unfortunately a bug was found in the implementation of mapping. The problem is that referred elements are always eagerly fetched. Specifying of fetching to be lazy does not work. This leads to useless loading of many data.

5.3.3 Data Access Layer

The data access layer is based on Hibernate `Session` interface. A Session is used to get a physical connection with the database. The main function of the Session is to offer create, read and delete operations for instances of mapped entity classes. Operations have to be surrounded with a transaction. At the beginning `beginTransaction()` has to be called on the session object. At the end of the transaction, `commit() is called`. If an exception is thrown by the `Session`, the transaction must be rolled back.

The Data Access Object (DAO) pattern was used for the data access layer. DAO is an object that provides an abstract interface to a database or other persistent storage. The advantage of using it is separation the two layers that can know nothing about each other. The class diagram with DAOs is depicted in image 5.5.

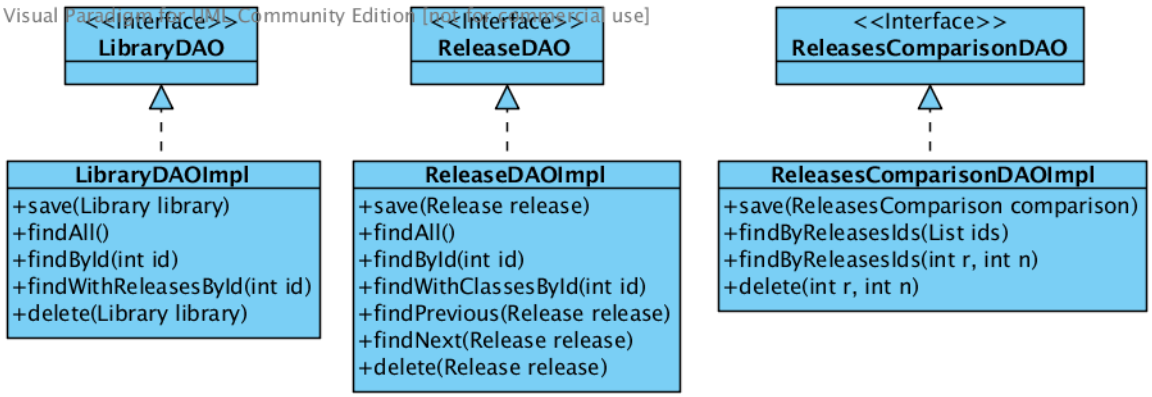


Figure 5.5: Data access objects.

For each meaningful entity a separated DAO was created, so the interfaces `LibraryDAO, ReleaseDAO, ReleasesComparisonDAO` were created. Implementations of the interfaces can differ depending on the used DBMS. Methods of the interfaces are contained only in the implementations for simplicity.

The contained implementations all work on Hibernate, which already provides an abstraction above different DBMS. However, it is not obligatory to use this abstraction. Hibernate enables three ways of querying by native SQL, by Hibernate Query Language (HQL)

or by Criteria API. HQL and Criteria API provide previously mentioned abstraction above different DBMS. HQL is similar to SQL, but HQL is object-oriented and understand notions like inheritance, polymorphism and association. Criteria is an alternative to HQL. Queries are build dynamically using an object-oriented API, rather than building query strings.

All DAOs contain `save` methods for saving or updating objects in the database. Methods `findAll` return all objects without any restrictions, whereas methods `findById` return only one object with the specified id. Methods `delete` remove an object from the database.

There are some special methods. `LibraryDAO` has the method `findWithReleasesById` for retrieving the library with its releases. Analogically, the method `findWithClassesById` in the `ReleaseDAO` returns the release with its classes. This DAO also contains `findPrevious` and `findNext` which retrieve the previous and the next release according to the date of a passed release. The DAO for comparisons provides two overloaded versions of the method `findByReleasesIds`. The first one, which requires a list of release ids, returns comparisons between these releases. The second one requires only two release ids and returns one comparison between the releases.

Creating a new connection to the database for each user may consume a lot of time. Connection pooling solves the problem. A number of shared database connections is maintained in a cache. Each time a user needs a connection, it is obtained from the pool. When the user does not need it, it is returned back to the pool. This enhances the performance of commands executed on the database. The DBCP connection pool was used in the developed system. Many parameters can be configured in this connection pool. For example the initial number of connections that are created when the pool is started, the maximum number of active/idle connections and the minimum number of idle connections in the pool. The default values were sufficient.

When the system runs a long time without user requests, an exception was thrown. This problem occurred because no opened connection was in the pool. Setting the minimal number of idle connections and the initial number of connections, which is zero by default, could be the solution. However, a better solution is to check whether the valid connection is in the pool, because the system is not used so often (at least at the beginning) to have some opened connections permanently. The `validationQuery` had to be configured to a select statement which returns at least one row, so `SELECT 1` was used.

5.3.4 Business Layer

All data access objects and `BCChecker` are encapsulated in the class `CheckerServiceImpl` that is an implementation of `CheckerService`. The class diagram of the service is on figure 5.6. Only methods from the implementation of the service are visible to simplify the image.

As you can notice from the names of service's methods, many of them are just encapsulated DAOs methods, although there are methods with more complicated business logic as well.

One of them is `createLibraryFromArtifact(Artifact artifact)`. This method cooperates with Maven Central repository through REST API. At first it obtains a list of all versions of the artifact. Then it downloads all the versions and stores them temporarily. It creates a library for the artifact and persists it in the database. The downloaded artifact versions are one by one parsed, stored in the database and removed from the disk.

The `parseAPI(Release release, File file)` uses `read(URI uri)` method to parse a file to release.

The system should provide compatibility overview of a library which means compar-

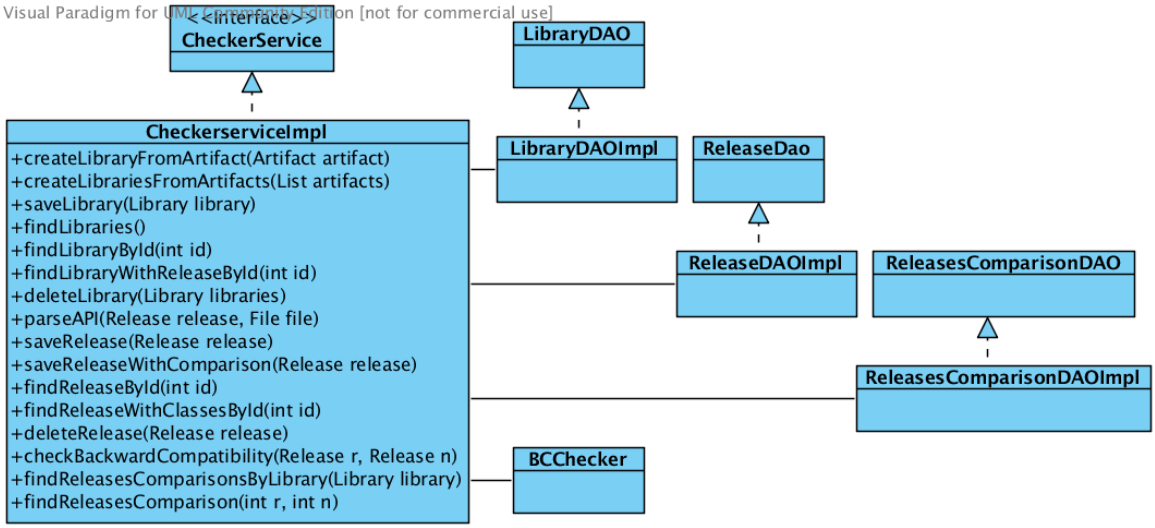


Figure 5.6: The service class encapsulates DAOs.

isons across the releases as they were released. So the comparisons between the releases could be stored in the database instead of their computation which takes some time. The method `saveReleaseWithComparison(Release release)` is exactly doing that. It uses `ReleaseDAO`'s methods `findPrevious` and `findNext` to find the previous and the next release. If the previous release or the next one is in the database, the comparison between the found release and the currently stored release is computed and stored. If both the previous and the next release are found (the release is inserted between them), then the comparison between them is removed.

Analogically to the method for creating a release, the method `deleteRelease(Release release)` removes the comparison when removing the release and it creates a new comparison. The new comparison is created only if the removed release had the previous and the next release.

The method `findReleasesComparison(int referenceId, int newId)` tries to find a precomputed comparison in the database. In case of failure it computes a comparison and returns it.

A layer of controllers is one level above the previously described service layer. A controller provides access to the application behavior defined by service layer and interprets user input and transforms it into a model that is presented to the user by the view. Controllers are defined by annotations in Spring framework.

```
@Controller
class LibraryController {

    @RequestMapping(value = "/admin/libraries", method = RequestMethod.GET)
    public String showLibraries(Model model) {
        Collection<Library> results = this.checkerService.findLibraries();
        model.addAttribute("libraries", results);
        return "libraries/list";
    }
}
```

The `@Controller` annotation indicates that a class has a role of controller. Requests are mapped to methods with `@RequestMapping` annotation. The method name is optional. The mapping parameter `value` serves for defining of URL and the `method` parameter is used to define an HTTP method of request (most commonly one of GET, POST, PUT or DELETE). The `model` parameter of the method is used to pass data to the view. The mapped method returns the path to the JSP file that will be showed as the view.

Four controllers were created in the system, see figure 5.7. The first two controllers handle ordinary user requests: `AccessController` that manages requests for logging in and logging out and `CheckerController` that handles requests for showing libraries, their compatibility overviews and checking of releases' compatibility. The second two controllers are used to arrange administrator requests: `LibraryController` for managing libraries and `ReleaseController` that is used for management of releases.

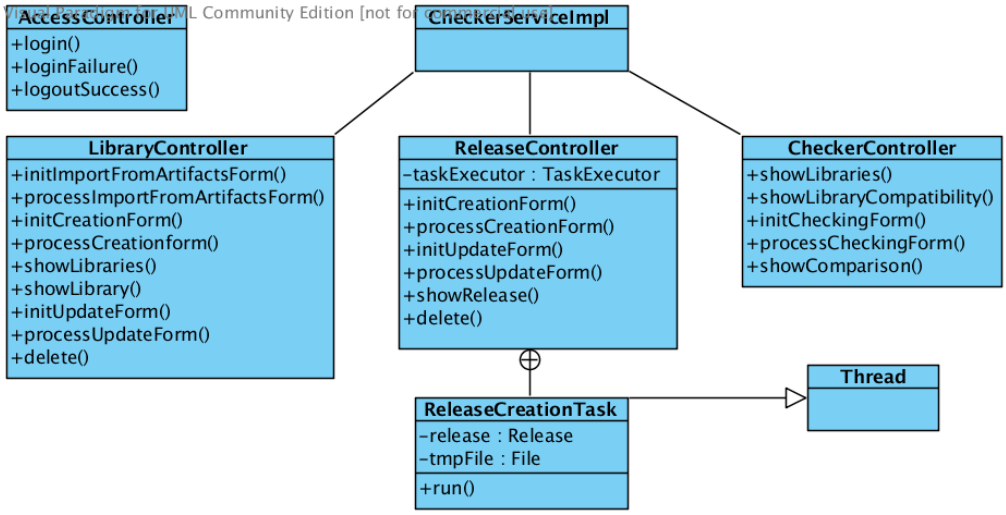


Figure 5.7: The controllers use the service.

Methods on the figure lack arguments to clarify it. Methods named `initXXXForm` are used to initialize forms and mapped to GET requests. Storing of objects in the database is done by methods named `processCreationForm` that are mapped to POST requests. Updating of database records is handled by methods `processUpdateForm`. These methods are mapped to PUT requests. Methods prefixed with `show` are intended to present data and mapped to GET request. Eventually, `delete` methods are mapped to DELETE request and they execute deletion of records from the database.

Method `processCreationForm` in `ReleaseController` is used to create a new release from an uploaded JAR file. At first, the file is stored temporarily. Then, parsing and storing of release in thread is processed, so the user does not have to wait on page and can continue working. The dummy release is stored in the database to make the user feel like the release was added immediately. The thread processing is implemented with the use of Spring's `TaskExecutor` that provides abstractions for asynchronous execution of tasks. The executor executes each submitted task using one of several pooled threads. It has a method `execute` that requires the task to execute. The class for a task was named `ReleaseCreationTask` and it inherits from `Thread` class. The class contains an implementation of `run` method where parsing of a file, storing in the database and removing the temporary file is done. In case of failure during creating a new release, the dummy release is deleted as well as a

temporary file.

Authorization and authentication is implemented with the use of Spring Security framework. All the configuration is in the file `security-config.xml`.

The system needs only one authenticated user – an administrator, other users accesses the system without logging in. The administrator's configuration includes credentials and a role – `ROLE_ADMIN`. A password is hashed with SHA algorithm, but it can be easily changed to another one. The hash of the password is stored directly in the configuration file that is not accessible through the internet due to the restrictions of a web container.

Authorization is also configured in the same configuration file as authentication. Authorization is set to allow access to all resources (images, CSS, JavaScript, etc.) and all pages except URLs prefixed with `/admin/`. All pages behind these URLs are protected with authentication.

5.3.5 Presentation Layer

The presentation layer is based on Java Server Pages (JSP). JSP is high-level abstraction of Java servlets. It is translated into the servlets at runtime. JSP allows to define HTML in combination with Java code. It supports Expression Language (EL), used to access data and functions in Java objects. An example of EL for obtaining a value of property `name` from object `library` is `${library.name}`. It is possible to compose a web content by importing one JSP to another using `<jsp:include page=‘‘fragment.js‘‘/>`. It was used for importing menu or head tags to the page.

Writing an extensive Java code in JSP might become confusing. Thus a classical technique called *scriptlet* was not used: `<% ... %>`. I have used Java Standard Tag Library (JSTL) instead, which is a JSP tag library. JSTL has support for common, structural tasks such as iteration and conditionals. The examples of tags are `<c:out>` for printing evaluated expressions, `<c:if>` for conditional evaluation of body if the supplied condition is true and `<c:choose>` for mutually exclusive conditions.

Except the JSTL, Spring and Spring form tag libraries were also used. The `url` tag was used from Spring library. It creates URLs with support for URI template variables and HTML/XML escaping. The Spring-Form library comes with form and input tags with data binding ability. This form tag library is integrated in Spring Web MVC, giving the tags access to the command object and reference data a controller deals with. Tags used from the form tag library are for example: `form`, `input`, `select`, `errors`.

Spring Security Tag Library served for checking whether user has a particular role, so that the page content could be customized. The tag with the ability is `authorize`. It was used for displaying menu links according to the role.

JavaScript and jQuery were used to provide good user experience. For example, when a user displays a form, the first input is focused. If the user selects a file to be uploaded when creating a release, the name of the file is automatically filled. A component `datepicker` from jQuery UI was used for selecting a date in an interactive calendar showing on focus to date input.

Furthermore, Dandelion DataTables component was used. This component allows to use jQuery's DataTables component in Java/JEE based applications. It offers features like sorting columns, searching a table, column filtering and styling.

Chapter 6

Experimentation and Optimization

It was experimented with the system with the purpose to optimize time and space complexity. The experimentation was done in the system with the following configuration: Intel i5-254M 2.6 GHz 64-bit architecture, 8 GB RAM, Fedora 18, kernel 3.8.11-200.fc18, Java 1.7.0_19 (OpenJDK), MySQL 5.5.31. The first thing what had to be considered was the input data. The following libraries were chosen from Maven repository:

GroupId	**ArtifactId**	**Count of versions**	**Size [MB]**
commons-lang	commons-lang	12	2.1
joda-time	joda-time	16	8.5
org.javassist	javassist	6	4.0
junit	junit	20	3.7
org.codehaus.plexus	plexus-utils	52	11.5
org.slf4j	slf4j-api	39	0.8
		145	30.6

All the libraries are intended for different purposes, created by various developers and of different size.

The database size in all the experiments was measured with variations of the following SQL select:

```
SELECT table_schema "Data Base Name",
sum( data_length + index_length ) / 1024 / 1024 "Data Base Size in MB",
FROM information_schema.TABLES
WHERE table_schema = "japi-checker-web"
GROUP BY table_schema;
```

Time consumption was measured with the Java method `System.nanoTime()`, that was put around the measured block of code (typically a method body).

```
long start = System.nanoTime();
// measured block of code
double elapsedTimeInSec = (System.nanoTime() - start) * 1.0e-9;
```

6.1 Visibility Limited Parsing

The first experiment is used to measure the space of data in the database and the time of parsing and storing of releases. The database was empty before the experiment. However, it still took a space of 0.45 MB. Except the size of the whole database, the size of elements (tables class, method, field, etc.) and the size of comparisons (tables release_comparison, difference, argument) was measured. Comparisons between releases in the order as were released were stored. That for example means that 11 comparisons were stored for commons-lang that contains 12 releases. The time of parsing (service's methods `parseAPI` and storing `saveReleaseWithComparison`) of releases was also measured.

Stored elements	DB [MB]	Elem. [MB]	Comp. [MB]	Parse [s]	Save [s]
All elements	31.97	30.81	1.06	2.04	308.71
API elements	22.27	21.17	1.00	1.53	215.76

If all Java elements are stored in the database, it takes even more space than JAR files. Storing only API elements means 31.29% reduction of the memory. Visibility limited parsing was considered as very useful, so it will be present in the system in the following experiments.

6.2 Loading vs. Computing of Comparisons

The goal of this experiment is to find what time is needed for loading of precomputed comparisons and for their computation on demand that comprises loading of releases and computation of their comparison.

The bug in *any* mapping described in the section 5.3.2 causes loading of elements associated to differences. The bug would have influence on the result of the experiment, so it was resolved by commenting out *any* mapping in the Hibernate mapping file. As a consequence, the source file of change is not present in the comparison report.

Several pairs of releases were chosen for the experimentation. These were mostly pairs with high number of differences because of low time needed for computing.

Reference	New	Load. C. [s]	Load. R.	Comp. C. [s]	Diff.
commons-lang-2.3	2.4	0.0714	3.66	0.0283	133
javassist-3.16.1-GA	3.17.0-GA	0.2294	5.41	0.0378	361
javassist-3.14.0-GA	3.15.0-GA	0.0022	5.22	0.0160	2
joda-time-1.6.2	2.0	0.1252	6.16	0.0301	200
junit-4.9	4.10	0.0059	2.33	0.0189	11

Loading of the precomputed comparisons is much faster than computation of them. The time of loading depends on the number of differences. The average loading speed of differences is 1301.22 differences/s. The average computing speed of differences is 23.28 differences/s. However, this speed does not depends to much on the number of differences. It rather depends on the size of API elements that have to be loaded before comparison. So loading of precomputed comparisons is at least 55.89 times faster. Actually this number would be much higher, but pairs with high number of differences were mostly chosen for the experiment. That means storing of comparisons is very desired.

6.3 Choosing of Comparisons to Store

Storing of all comparisons is the most efficient approach how to obtain comparisons in terms of time. The question is what space is required for doing of that.

The average space needed for storing of one release is equal to fraction of total size of comparisons and number of comparisons. If we rely on the data measured in the experiment described in section 6.1 the average space needed for storing of release is $\frac{1.00}{139} = 0.0072$ MB.

Count of all comparisons can be computed:

$$\sum_{i=1}^{lc} P(rc_i, 2) = \frac{12!}{(12-2)!} + \frac{16!}{(16-2)!} + \frac{6!}{(6-2)!} + \frac{20!}{(20-2)!} + \frac{39!}{(39-2)!} + \frac{52!}{(52-2)!} = 4916,$$

where lc is count of libraries and rc is count of releases. So the memory needed for storing of all comparisons is $4916 * 0.0072 = 35.39$ MB. It can be reduced to approximately half (17.70 MB) by storing of backward compatibility comparisons only, as was described in the section 4.3.2. This number is even much lower than space needed for storing of API elements which is 21.17. The real number would be with high probability much more higher, since releases between that several releases were released differ in API more than a release from the previous release (comparisons between these releases were used in the calculation).

Algorithm choosing releases for creating of backward comparisons was implemented for this experiment, with the motivation of achieving relatively low memory requirements of comparisons, advised by the previous calculation. So the real memory consumption can be measured and consequently validated the calculated one. The following table contains total size of the database, size of API elements, size of comparisons, count of comparisons and time required for storing of the data.

Choosing alg.	**DB [MB]**	**Elem. [MB]**	**Comp. [MB]**	**Comp.**	**Save [s]**
Linear comp.	22.27	21.17	1.00	139	215.76
Backward comp.	56.61	21.17	35.34	2458	345.45

As we have expected the real size of comparisons is higher than the calculated one, two times higher. In case of storing backward comparisons is needed 35.34 times more space for comparisons than in case of storing linear comparisons. Choosing algorithm has to be considered in dependence on the available means whom the production environment disposes. The algorithm choosing releases linearly was set as default.

Chapter 7

Requirements and Deployment

The library and the server system requires Java compiler and Maven for building. Both of them require Java for running. The server system additionally needs MySQL Server.

The final system was deployed to OpenShift. The demonstration can be found on the link: https://japichecker-trohovsky.rhcloud.com. OpenShift is a cloud computing platform as a service (PaaS) product from Red Hat. The service runs on the open-source software OpenShift Origin. Git can be used to deploy web applications. The application is deployed every time when changes are pushed to the Git repository. Besides Java, the service supports also Perl, PHP, Python, Ruby and Node.js. Supported databases are MySQL, PostgreSQL, MongoDB. OpenShift provides both automatic and manual scaling of the resources by adding instances of the application. The limits per application are 1 GB of storage and 512 MB of memory. Which is quite low for a production version, but it is enough for the demonstration.

It is possible to use source code deployment or deployment of precompiled project. When a Java source code is pushed to the repository, it is automatically built with Maven and the compiled WAR file is deployed to the web server. In case of precompiled project deployment, the WAR file is pushed to the repository. I have tried both, but WAR deployment was more suitable because of the need to include a modified japi-checker, which is still not in the Maven repository, so in the case of source code deployment it had to be installed to the local Maven repository.

Chapter 8

Conclusion

Changes causing incompatibility on an API and an ABI level were analyzed and described in the section 2.3. Less evident changes were practically verified. Existing solutions for detecting API and ABI compatibility were analyzed. The tools were tested with the purpose of finding which detection of incompatible changes is supported.

One of the analyzed tools was chosen – japi-checker. Extensions and improvements of this library were proposed. I have implemented a basic CLI which had been missing. The API model and parsing of bytecode were improved as well as compatibility detection. The library's functionality was extended to distinguish between source and binary compatibility. Missing detection of incompatible changes was implemented. It includes incompatibilities caused by adding a field or a method, changing field's final attribute or value, making a method abstract or changing its value, changing method's variable arity parameter to an array type. Furthermore, detection of incompatibilities related to type parameters was provided. Already implemented rules were revised and in some cases changed or entirely replaced. An example is detection of inheritance changes or changes of API type.

The server application which provides information about incompatibilities was designed and implemented. The server application is based on previously implemented library for detection of incompatibilities, Hibernate and Spring framework. The algorithm of balancing space and time complexity was suggested in analysis and implemented. Maven Central repository is used as the source of libraries. They can be imported through a web interface.

The final server application was experimented. Three experiments were done with the goal to measure space and time complexity and optimize them. The first experiment showed the advantage of storing only API elements, space requirements are then reduced by 31.29%. The second experiment showed that loading of precomputed comparisons is 55.89 times faster than its computation on demand. The result of the previous experiment leads to implementing of an algorithm that stores 50% comparisons. The system was experimented with presence of the algorithm. The size of stored comparisons was 35.34 times higher and it was even 1.67 times higher than the size of stored API elements. That is quite high, so the previous algorithm was set as default.

The library, or more specifically the CLI build above the library, has already found its users. The server application is still in the phase of demonstration because of high memory requirements and absence of relevant free hosting, but it has a potential to become the place where Java developers will first come when they want to upgrade libraries they use.

8.1 Future extensions

Many future extensions can be done in the library and the server application. The library can be extended to be able to detect compatibility with knowledge of a client code. This feature can also be used in the server application. Data in the server application are currently filled manually by an administrator. When the new version of tracked library is released, the administrator has to add the new release manually. Scanning of data source of libraries (Maven Central) and automatic downloading of newly released versions of libraries would solve the problem. Another very useful extension is providing the server system API to other systems, for example with using REST.

The server application could be tested with the use of some other DBMS like PostgreSQL. It would also be interesting to try some NoSQL database. It could be an optimization of time complexity when storing a big amount of data. Having parsed APIs of many libraries gives an opportunity to find some useful data about how the APIs are designed. Some advanced data analysis can be done above the stored APIs.

Bibliography

[1] Jim de Riviers. Evolving Java-based APIs. Retrieved January 2, 2013, from http://wiki.eclipse.org/Evolving_Java-based_APIs, 2009.

[2] James Gosling et al. The Java Language Specification. Chapter 13. Binary Compatibility. Retrieved November 2, 2012, from http://docs.oracle.com/javase/specs/jls/se7/html/jls-13.html, 2012.

[3] Joseph D. Darcy. Kinds of Compatibility: Source, Binary, and Behavioral. Retrieved January 3, 2013, from https://blogs.oracle.com/darcy/entry/kinds_of_compatibility, 2008.

[4] Oracle and/or its affiliates. Sigtest User's Guide. Retrieved January 1, 2013, from http://docs.oracle.com/javame/test-tools/sigtest/2_2/html/index.html, 2011.

[5] Sonatype Inc. Maven Central Repository. Retrieved May 18, 2013, from http://search.maven.org/, 2013.

[6] Dennis Sosnoski. Classworking toolkit: Generics with ASM. Retrieved May 15, 2013, from http://www.ibm.com/developerworks/java/library/j-cwt02076/index.html, 2006.

[7] Eric Bruneton. ASM 4.0 A Java bytecode engineering library. Retrieved March 15, 2013, from http://download.forge.objectweb.org/asm/asm4-guide.pdf, 2006.

[8] Rod Johnson et al. Spring Framework Reference Documentation. Retrieved May 17, 2013, from http://static.springsource.org/spring/docs/3.2.x/spring-framework-reference/htmlsingle/, 2013.

[9] The Hibernate Team. Hibernate Reference Documentation. Retrieved May 17, 2013, from http://docs.jboss.org/hibernate/orm/4.2/manual/en-US/html_single/, 2013.